Karlsruhe Institute of Technology

Interactive rendering of large and unstructured particles

Bachelor Thesis of

Albert Pérez Toro

An der Fakultät für Informatik
Institut für Visualisierung und Datenanalyse,
Lehrstuhl für Computergrafik

September 18, 2017

Reviewer:	Prof. Dr.-Ing. Dachsbacher
Second reviewer:	Prof. Dr. Prautzsch
Advisor:	Tobias Rapp

Abstract

Nowadays interactive rendering of large and unstructured data is turning into a normal practice in the scientific community. This kind of data can be visualized with standard API's like OpenGL, but these API's are not the most efficient option, since are based on processing triangle meshes and point clouds are very dense for this practice. Furthermore, the manufacturer does not allow to modify hardware details to correct this fact. For these reasons, in this paper we present a point cloud rendering pipeline capable of processing large and unstructured data more efficient and faster. Moreover, a comparative is shown between the standard API's pipeline and our point cloud pipeline, which demonstrate that standard API's pipeline should be given up for this kind of data.

Contents

1. Introduction

Nowadays computer graphics is gaining importance in our society, computer graphics are the responsible of representing data into an image, in an effectively and meaningfully manner to the user. With computer graphics we can process image data from physical phenomenons or figures. There is no doubt of the influence and importance of computer graphics in the media, with animations, film effects, medical imagery, video games, movies...

Although it exists several fields in computer graphics, in this project we have focused on the field of rendering, more concretely, in the visualization of large and unstructured data, such as point clouds.

At the moment, scientific communities and institutions have the necessity to visualize data from diverse physical simulations or phenomenons,some of them require an interactive rendering to have a better understanding of the data.

Lot of physical simulations generate huge non-structured data. For instance, the Smoothed Particle Hydrodynamics (SPH) simulations, are simulations of the dynamics of continuum media, such as solid mechanics and fluid flows. In these kind of simulations, we can easily find millions of particles through the time with different attributes. These particles have to be represented and rendered as point clouds.

Another example can be Illustris project, which consists on a large cosmological simulation of galaxy formation using a state of the art numerical code and a comprehensive physical model. This project is also a project that need to process large and unstructured data as point clouds. As we have seen, many projects require the visualization of big data that is non-structured, since the data is non-structured, it should be treated as a point cloud.

There exists lot of API's like OpenGL specialized in the rendering of structured data with triangle primitives, using special techniques for this type of primitives, that accelerates the process of rendering. On the other hand, these API's are not the most efficient rendering non-structured data, because some techniques that are applied to this data are not worthy or useless. For this reason, significant efforts should be done to develop a new pipeline in order to render this data as efficient as possible. This optimized pipeline would be the most adequate way for this projects to render interactively their huge data. For unstructured data, we should work with points primitives and not triangles primitives, as OpenGL does, for point primitives we should abandon the standard API and directly switch to GPGPU API like OpenCL.

Moreover, the exchanging data between the CPU and the GPU needs to be controlled and keep it low, since it is a potential bottleneck in case of large data. We often see this problems in very demanding applications that requires lot of data to render, for example in computer games.

The main objective of this thesis is to develop and program a GPGPU rendering pipeline optimized for point clouds, including the steps of depth test, shading of particles and efficient, rasterization. This GPGPU pipeline should work faster than standard API's pipelines.

Furthermore, OpenCL will be used to obtain parallelism in the processes, making the pipeline more efficient.

In this document we will present the theoretical and practical knowledge to build this pipeline, the project consists on a time comparison between a triangular primitive pipeline and a point primitive pipeline using point clouds as data. We will present also the reasons why the point primitive pipeline is processing faster this type of large and unstructured data.

The point primitive pipeline as we have mentioned before, will be programmed mostly with OpenCL. First of all the data will be processed through the pipeline, creating an OpenCL image that will be transferred to OpenGL as a texture, after that, OpenGL will render that texture and we will see the data through the screen. In this paper we will explain every single step of this procedure, it is organized as follows: after the short overview and the related work, a theoretical knowledge about coordinates transformations will be explained. Afterwards our pipeline theory will be explained and after that we will explain the implementation of that pipeline. Finally, an evaluation section will describe and analyze the results of our experiments. The document will conclude with a conclusion, expressing the results and future to follow.

1.1 Motivation

There is no doubt of the importance of computer graphics in our society, as I have mentioned above. Also the management of point cloud visualization data is gaining importance for the different applications that use this type of primitives, such as simulations, laser scanning systems... In this document, we will see how to manage large and unstructured data and how they should be processed in the pipeline. Moreover, we will show the processes of a triangle primitive pipeline that are unneeded in a point primitive pipeline, and the reasons for it.

OpenCL will be the programming language that we will use to program the GPGPU pipeline, this programming language will allow us to parall the different processes. OpenCL is a new language, it was launched for the first time in 2009, so it will be an interesting fact, to work with new technologies in the project.
Furthermore, in some steps of our program, OpenCL and OpenGL need to interoperate and this can be a challenging task, in this document we will tackle the problem, offering a practical solution to this problem.

Another motivation of this project, is the challenge to avoid race conditions in the different kernels of OpenCL when they are working in the same resource, in this document we will also offer a practical solution to this problem. In this project we can differentiate diverse sectors of knowledge, first of all, the theoretic knowledge of the different steps of a pipeline, and how they should be implemented. Into this steps we should also include knowledge of coordinates transformations, necessary to implement any pipeline.
Another section of interest in this project are the different languages or API's used like OpenCL and OpenGL, they are really popular and used in computer graphics.

Different motivations push me to do this project, first of all as I have said learn new techniques and technologies, learning and applying theoretical and practical knowledge.

2. Related Work

During the development of this project we have consulted different reports and books that have help us to have a better understanding about computer graphics. Moreover, some books offered us solutions to different problems that we had.

The first report we had read and investigate, [GKLR13], in this report the author is showing a self-implemented point cloud rendering pipeline and compare the results with a standard API's pipeline. From this report we have subtracted different ideas and methodologies of how to organize and design a point cloud rendering pipeline. Moreover, the report offer different algorithms for the implementation of rasterization. Finally, the report shows a comparative between the point cloud rendering pipeline and the standard API's pipeline.

Since OpenCL is a relatively new programming language, [Tay13] has been a great help. This books explains the programming basics of OpenCL showing some programming examples and giving tips and best practices.

Another interesting area for this project, is the ability to work with kernel and parallel threads, since the processing of the data, to be processed faster, has been done by parallel threads in OpenCL. For this reason, a good source of information that had helped us to develop the project are the books [LW85, Tay13].

Paradoxically, another interesting area had been key in our project is the study of triangle primitive pipelines, studying this kind of pipeline that does not work efficiently with point clouds we could detect which parts were not efficient and have some ideas to our point-cloud pipeline. For this reason, we had take into account the reports [GKLR13, HA11], that as we have mentioned above, are studies that compare a specific cloud point pipeline with the general pipelines used for triangle primitives. As we can see in the reports, the point cloud pipeline work faster than the triangle primitive oriented pipeline. These reports has been key to understand the different reasons that lead to the point cloud pipeline to work faster.

Another interesting topic in our project was the different optimization techniques to create an efficient pipeline, to optimize our pipeline, in reports [GKLR13, HA11], some of them are shown. Also in [LW85] we can observe how to work with point as a primitive and the different problems that can happen. This report, also suggests different ideas for the point cloud pipeline. An interesting area will be the programming area and the language

programming involved, for this reason we will use different books and webpages to programm correctly with OpenCL [nvi, int, AMD, Sca12, Tay13, op]. More concretely for the interoperability processes between OpenCL and OpenGL we have worked as [int] had suggested us, the webpage presented three different ways to do this processes.

3. Transformations

In this section we will express our knowledge regarding the different coordinates transformations that suffers the vertices in the point cloud rendering pipeline. As we can see in 3.1, the vertices follow a strict process.

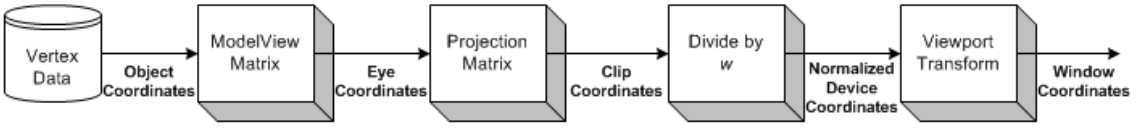

Figure 3.1: Coordinates tranformations

3.1 Object Coordinates

In the first instance we can observe the object coordinate system. In that coordinates, the designer or modeller should decide a point that will be the origin of the object, and from this point basically start modelling the object. The designer also should set the orientation of the object and the set of model axes.

3.2 Eye Coordinates

The eye coordinate system, also known as camera coordinates or view coordinates, is the system coordinates where points or vertices vary depending their position and the position of the camera. In this coordinates system, the camera will be always situated in the origin (0,0,0). We can see a clarifying example in Figure 3.2.

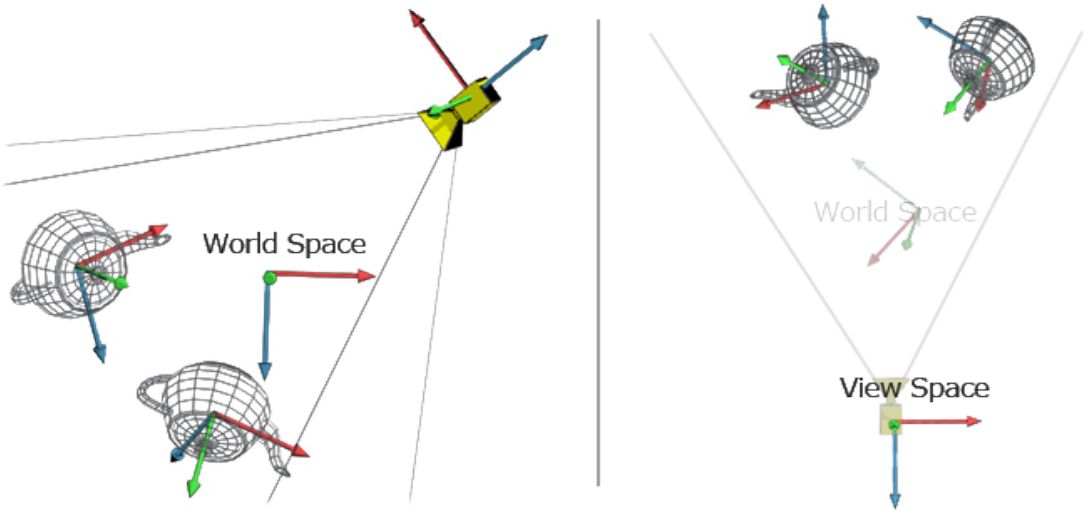

Figure 3.2: World and Eye Coordinates

To convert a point from object coordinates to eye coordinates we need to multiply the coordinates of the point per the View Matrix. Since the point coordinates have only 3 coordinates and the view matrix is a 4x4 matrix, a fourth coordinate (with value 1) should be added in the point. In this way we can obtain the coordinates of the point in eye coordinates.

To obtain the view matrix we will need to know different features of the camera as the position of the camera, the view reference point (point where the camera is looking at) and the up vector (vector that describes the "up" direction of the camera).

In OpenGL you can obtain easily the View Matrix with the command *LookAt*.

3.3 Clip Coordinates

The clip coordinates are the eye coordinated multiplied by the Projection matrix, meaning that this coordinates take into account which portions of the objects will be visible to the user.

The Projection matrix express the frustum (viewing volume) of the user and how the data is projected on the screen. To clip a vertex basically the coordinates (x,y,z) should be in the range [-w,+w].

3.4 Normalized Device Coordinates

The normalized Device Coordinates are obtained from dividing each clip coordinate (x,y,z) by the homogeneous coordinate w. The Normalized Device Coordinates go from -1 to 1 since the vertices coordinates in Clip Coordinates were between the range [-w,+w]. The coordinates are more likely to the screen coordinates, but they still have not been scaled and translated in the screen.

3.5 Window Coordinates

The window coordinates are obtained from multiplying the Normalized Device Coordinates (NDC) with the viewport transformation. With this operation we are basically scaling and translating in relation to the rendering screen. This coordinates will be the ones passed

to the rasterization process. The information that we basically need to know to transform from NDC to Window Coordinates is the width and height in pixels of the rendering screen.

4. Pipeline Theory

The rendering pipeline is the sequential steps that are made to render an object, in this project we have divided this process into five sequential steps (Vertex Shader, Culling, Fragment Shader and Fragment Merging). Those steps need to be followed in order.

First of all, before describing the pipeline, we will have to define what is a shader. A shader is a program, that can be compiled independently, in charge of altering the color of an object/surface/polygon in the 3D scene, based on things such as the surface's angle to lights, its distance from lights, its angle to the camera and material properties. Shaders are found in the rendering processes and the principal objectives of shaders are to create a photorealistic effects.

Some of this steps contain parallel processes that has been programmed with OpenCL, that we will comment in the implementation section, in order to accelerate the process of visualization and making it more efficient. Since our pipeline is specially dedicated to render point clouds, some steps are differentiated from the traditional OpenGL pipeline.

For instance, in the cloud points pipeline, since we are working with zero-dimensional-primitives, we can omit the tessellation or geometry shader stage. Also, the normal array is not useful in the point cloud primitives, so we should exclude this information. In Figure 4.1 we can see observe the pipeline processing that we are going to implement:

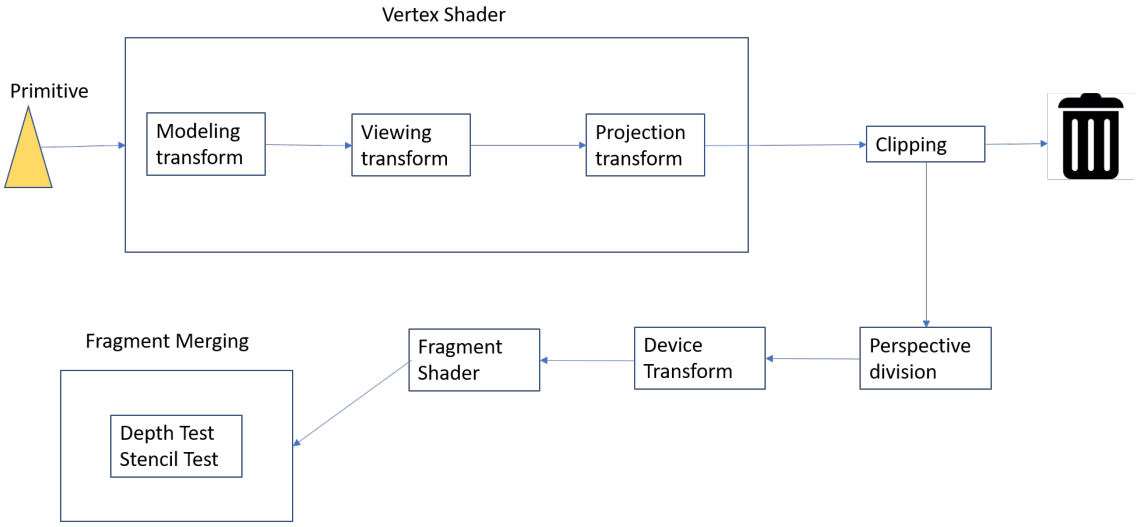

Figure 4.1: Render Pipeline

4.1 Vertex Shader

4.1.1 Objective:

A vertex shader is a shader that basically works with the structure of the vertices of the figures and the principal objective is to determine different attributes of themselves. Vertex shaders can not create new points, although they can modify attributes as position, color and texture coordinates. Vertex shaders can enable different effects to figures as movement or lighting

4.1.2 Inputs:

The user can modify the different inputs of this shader, but the principal of them are:

- Position of the different vertices: The different positions are of the different vertices, these are expressed in world space.
- View matrix: Matrix that will allow the shader to convert the points in world space to view space.
- Perspective matrix: Matrix that will allow the shader to convert the points in view space to perspective space.
- Normal vector: Vector perpendicular to a surface of the primitive.
- Normal matrix: Matrix which preserves vertex normal under an affine transform.
- Lighting matrices: Different matrices related with the lighting atmosphere, position, shininess...

4.1.3 Outputs:

Also the user can modify them but the most usual outputs are:

- Position: position of the particle, after applying the view and perspective transformations.
- Normal: Normal array after applying the different transformations.

4.1.4 Point cloud pipeline:

Our vertex shader will calculate the position of the vertices depending on the position of the camera and taking into account the perspective.
In this step we can find differences between our pipeline and the vertex shader of a triangle based pipeline. The first difference is that we don't work with the normal vector attribute and normal matrix, useful to know the orientation of the triangle primitive, but useless in a point primitive pipeline, since we don't need the orientation.
Since in this project the different simulations does not require any type of lighting, the different lighting matrices are also omitted.

4.2 Culling

4.2.1 Objective:

The culling process is the process of discarding the different vertices that are out of the perspective view with the purpose of saving time and energy.
Basically the vertices out of the perspective view are not visible for the user, for this reason they should not appear on screen and they should be discarded. Discarding this points will save time and energy, because these points won't be processed anymore by the pipeline, allowing the pipeline only to process the useful vertices and not all the vertices.
To know if a point is inside the perspective view, we have to verify that the first three coordinates (x,y,z) are inside the range of the negative value and the positive value of the independent coordinate. In case, some of the coordinates are outside this range, the vertex should be discarded.

4.2.2 Inputs:

- Vertices: The vertices are expressed in homogeneous coordinates, because in previous steps we had multiply the coordinates by the perspective and view matrix.

4.2.3 Outputs:

The output of this step consists of those input vertices that are inside the perspective view.

4.2.4 Point cloud pipeline:

In our pipeline, this step is included inside the vertex shader, for efficiency purposes. This step has no differences between the triangle primitive based pipeline and our pipeline, both has the need to discard this points that are outside the perspective view.

4.3 Fragment Shader

4.3.1 Objective:

The Fragment Shader is a program that goes through the different pixels of the object and calculate and assign a color for each pixel. The color can be calculated knowing different values, for instance, the different lights, normal vector, textures, properties of the materials... Also fragment shaders are allowed to calculate the depth and stencil values.
The depth values are stored in a depth buffer, where the nearest-Z is stored, the stencil values are extra integer values, per pixel, in the simplest case it stores the limit of area of rendering.
Fragment shaders can be an optional computation, when there is no fragment shader the color values are undefined and the depth and stencil values for the output fragments have the same values as the inputs.

4.3.2 Inputs:

Usually the inputs of this shader are:

- Front facing: It is related with the normal array, and basically shows the orientation of the primitive.
- Fragment coordinates: This attributes are the coordinates of the fragment.
- Primitive ID: This variables is an output of the geometry shader which relates the primitive with a geometric figure.

4.3.3 Outputs:

- Color Buffer: A buffer where we can store the colors of the different vertices.

4.3.4 Point cloud pipeline:

There is not a significant change between the fragment shader of a triangle primitive pipeline and a fragment shader of a point cloud pipeline.
In our pipeline the color of the pixel depends on the density of the particle. For this reason, we will only need the information of the density for each vertex. A different color will correspond to a different interval of density and then we will assign a color for each particle depending on which interval pertains the density.

4.4 Fragment Merging

4.4.1 Objective:

Fragment merging is the process of painting into the screen the different primitives , that were not discarded, and had successfully passed the different tests (depth test, stencil test...). In this step, the different tests play an important role, since particles that has been discarded by those test should not be painted. For example, particles with the same 2D-coordinates but different depths should be taken into account and the particle with greater depth should be discarded and not painted. For this reason, depth should be considered in this step, as well as stencil values.

4.4.2 Inputs:

- Color Buffer: A buffer where the colors of the different vertices are stored.
- Depth Buffer: A buffer where for each 2D possible point of the Z-nearest is saved.
- Vertices: The position of the different vertices.
- Stencil Buffer: Buffer where the stencil values are stored.

4.4.3 Outputs:

We can find an image as an output, where the vertices are drawn with the corresponding color that is stored in the color buffer.

4.4.4 Point cloud pipeline:

Since in our pipeline we are not using stencil values, we are not taking into account the stencil buffer and, for efficiency purposes, the depth test is done in this stage. Everything else is working also for the point cloud pipeline.

5. Implementation

In this section we will explain how the program was implemented, taking into account the pipeline theory and the different coordinates transformations. The principal idea of the program is to render the data given in an OpenGL texture. Previously to that rendering, the data has been processed by the point cloud pipeline and shaped in an OpenCL image that will be transferred to OpenGL as the texture we mentioned before. For this reason, one of the first thing we have implemented was the interoperability layer between OpenCL and OpenGL.

5.1 Interoperability OpenCL and OpenGL

As I have mentioned in the *Related Work* section we obtained various solutions for the OpenGL and OpenCL in the NVIDIA official webpage. The three of them consists on sharing an OpenGL texture but with three different commands (*clCreateFromGLTexture*, *clCreateFromGLBuffer*, *glMapBuffer*). The sharing texture with the *clCreateFromGL-Texture* command was the option we implemented and was the most recommended from NVIDIA official since it is the most efficient. The three methods of them are based on creating an OpenCL structure from an OpenGL texture.
Our Interoperability layer consists on creating first an OpenGL texture and then from this texture create an OpenCL image with the command *clCreateFromGLTexture*. Then we acquire the ownership of the OpenGL texture with the command *clEnqueueAcquireGLObjects*. After that the point cloud pipeline starts and when it is finished, the ownership of the OpenGL texture is released with the command *clEnqueueReleaseGLObjects*. After this process, the OpenGL texture have captured the data from the OpenCL image.

5.2 Vertices attributes

During the implementation of this pipeline, we have adapted the pipeline to the scientific data from the Ilustris project. This data is large and unstructured scientific data and will be our reference data. This data consists of 3 coordinates and a density, additionally, for testing purposes we have added a fifth value, that refers to the particle size. The density of a particle will be represented on the visualization by colouring the particle depending on the value of density.

The fifth attribute, given by the data, will consist on the particle size. This value will always be equal or grater than zero and will express how many layers of subparticles

surround the central subparticle. As we can see the picture when the particle has size 0, it is formed only of the central subparticle, when it is 1 it is formed by 1 layer of subparticles around the central subparticle. A clarifying example is shown in 5.1 where the particle size 2, and each cell represents a pixel of the screen space.

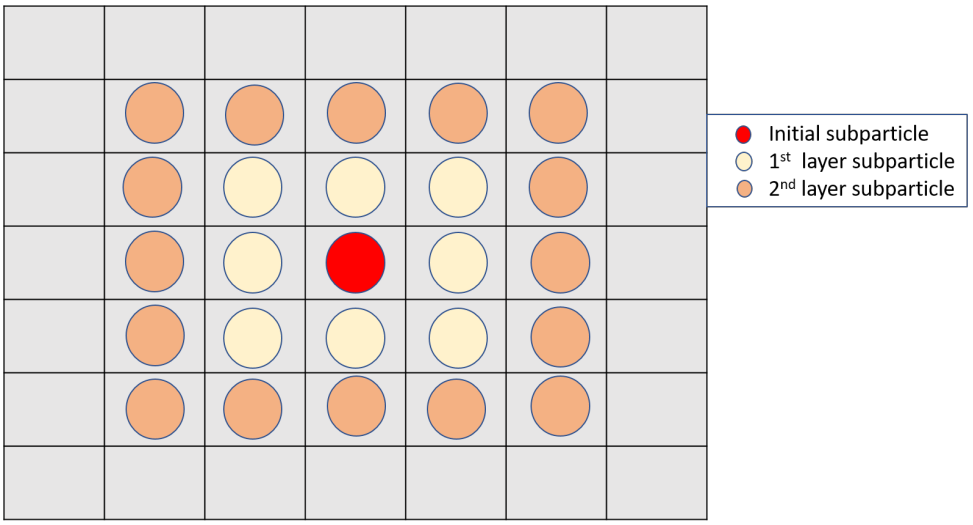

Figure 5.1: Particle size attribute example

5.3 Important structures and variables used

During the development of the program we have chosen different structures, the first structure that we chose was an image for the interoperability of between OpenCL and OpenGL, conceptually was easier to compare an OpenGL texture with an OpenCL image than the other structures, also more efficient for painting. Other structures that we could choose was a buffer or a map, but were less efficient in the interoperability.

For saving the different vertices we had choose an OpenCL buffer, each vertex occupy 3 contiguous positions, (x,y,z) respectively, we could have created 3 different buffers for each coordinate, but creating only one OpenCL buffer for the vertices was easier to send the vertices positions to the kernel.

Moreover, the density buffer is another buffer, we could add the density as a fourth coordinate in the previous buffer, but this would have make the program less maintainable and more complicated to understand. The first position of the density buffer corresponds to the density of the first vertex in the vertex memory buffer.

Furthermore, the depth buffer could have been an image or a matrix, that conceptually seems easier since each cell would represent a position of the OpenCL image that contain the draw. On the other hand, the choice was of being an OpenCL buffer was technical, since when an OpenCL image is passed on a kernel, only can be write or read, not both, and that was an important requirement of our program. For this reason we chose the depth buffer to be an OpenCL buffer memory.

5.4 Vertex Shader

We have implemented the vertex shader as a kernel function in order to parallelize the process. The vertex shader is transforming the vertices coordinates from world space to screen space, making the different multiplications and applying the clipping process. The

Vertex Shader will receive 2 buffers, one will be an input buffer with the vertices coordinates and an output buffer to save the output vertices. For efficient and speed purposes, the clipping process will consist on assigning the values -INTMAX to the coordinates of those vertices that are out of the frustum of view, then in the fragment merging we will take into account this situation.

The most efficient way to parallelize this process is creating different threads, but how many threads should we create? Since what we want to do is process all the vertices and there is not any dependence, the best option is to create a thread for each vertex, and then execute the kernel function for each thread. In that way all the different vertices are being processed in parallel.

5.5 Fragment Shader

We have implemented the fragment shader as a kernel function in order to parallelize the process. In the fragment shader basically we are assigning colors to the Color Buffer depending on the density, the first position of the color Buffer will correspond to the color of the first vertex. Since what we are trying is to process the Density Buffer, a thread will be created for every vertex, so all he positions of the Density buffer will be process at the same time.

5.6 Fragment Merging

We have implemented the fragment merging as a kernel function in order to parallelize the process, since we want to process all the vertices in parallel, the number of threads will be the number of vertices. During the Fragment Merging we will have to take into account three main factors, the depth test (including race condition problem), the size of the particle and if the particle is out of frustum.

If the particle is out of frustum the coordinates are -INTMAX, as we have caluclated in the vertex shading process, so basically we have to ensure that the coordinates of the particle are positive.

For the depth test we will have to compare the depth of the vertex and the depth buffer, if the particle seems to be near than the depth buffer then the particle is drawn and the depth buffer is updated with the depth of the particle. Since other threads are working in parallel the main challenge is to avoid the race condition problem, because it can be possible that two threads access at the same time to the same position of the depth buffer and updating their results writing a wrong Z. For this reason, to avoid ace condition, we have used the commands *atomicmin()* , to ensure an atomic access to the depth buffer, and can't be possible that two different threads access at the same time this resource.

Regarding the particle size, the kernel function reads the size of the particle and try draw the neighbour subparticles that compose the particle if they pass successfully the depth test. In case a subparticle doesn't pass the depth test, it is not drawn. Another case a subparticle is not draw can be because it is out of the screen, then we should ensure that the subparticle is inside the range [0,ScreenHeight) and [0, ScreenWidth] for the coordinates y and x respectively.

The color of a particle is read from the color buffer and all the subparticles caused by the particle size have the same color.

5.7 Calculating times

To make the comparative between the standard API's pipeline, like OpenGL and our pipeline we had to calculate the time of the process of rendering. The first idea we got was using a *clock* with the library of C++, but this idea is incorrect, since is only timing the CPU and we are interested in timing the GPU. For this reason we have implemented in our code a query in OpenGL language, with the commands *glBeginQuery* and *glEndQuery* we are able to calculate the time that has transcurred in the GPU.

5.8 OpenGL features

We have implemented different features regarding the rendering in OpenGL, for example, we have implemented a feature that allow the user to navigate through the visualization and observe more in detail the point cloud. This feature consists on modificating the position of the camera when the keyboard letters "W","A","S","D" are pressed, allowing the user to move in the scene. Moreover, the user can zoom and unzoom the point cloud scrolling the mouse.

To offer the "walk around scene" service to the user, we have calculated the Euler degrees of the movement and apply to the variables View Referenece Point of the camera and the camera position.

6. Evaluation

As we have mentioned before, this section will be dedicated to evaluate the speed of the point cloud pipeline and the standard API's pipeline, in this case OpenGL. We have used different types of data, real data from Illustris project and also data created for testing purposes. There are different factors that can affect the comparisons between the OpenGL pipeline and our pipeline. One of this factors is the number of z-tests and the other factor is the size of the dataset.

For our experiments we have used as a CPU an *Intel Core i3* and for the GPU a *AMD Radeon Graphics*. The viewport used is the size of 1024 x 1024 pixels. The methodology followed was render the different datasets and take more than 20 samples times of the pipeline loop, which we considered enough to make comparisons.

6.0.1 Syntetic data

We have created many syntetic data for this project, each data has different purposes and the main goal is to evaluate different attributes or features and how affect the timing of both pipelines and make a comparison.

6.0.1.1 Dataset 1

The first dataset is a small dataset, consist on 300 different vertices and there is not any depth test conflict. The particle size of the particles are 0 to not cause possible depth conflicts. With this dataset we are trying to find out which pipeline is faster under conditions of no stress and withouth Z conflicts. We can observe the different times in the Figure 6.1.

As we can observe in Figure 6.1 the times produced by OpenGL are lower under this conditions, when the number of vertices is very low. This phenomena can happen because in our pipeline is designed to treat large number of vertices and all the chores of paralleling are not worthy when the dataset is small. Instead, OpenGL pipeline seems to work faster for small datasets. The average time for the point cloud pipeline is of 158.89 ms and for the OpenGL pipeline is 115,70 ms. In this situation the speed up of the point cloud respect the OpenGL pipeline is of 0.73.

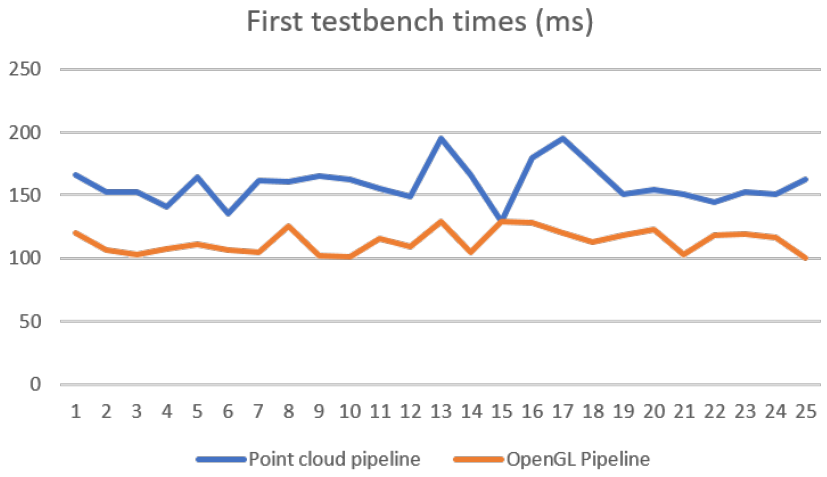

Figure 6.1: Plot of the different times of the first dataset

6.0.1.2 Dataset 2

The second dataset consist on a small dataset, 400 vertices, but the characteristic part of this vertices is that lot of them share the same x y coordinate, provoking lot of z-test conflicts. This dataset will be useful to measure the impact of the Z-tests in the rendering time in the pipelines. We can observe the results of this dataset in the next figure:

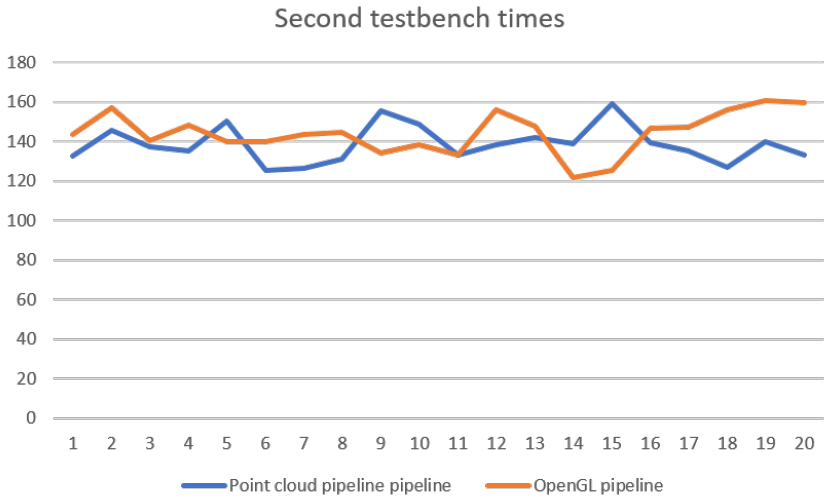

Figure 6.2: Plot of the different times of the second dataset

As we can observe in Figure 6.2 the times in this dataset are quite similar, the average time of the OpenGL pipeline is of 144.02 ms and the average time of the point cloud pipeline is about 138.59 ms. The point cloud pipeline is processing the information faster in this dataset comparing to the OpenGL pipeline, although the difference is very little and we can consider it insignificant.

Surprisingly, the point cloud pipeline seems to work a little bit faster in this dataset than the in first dataset, although the difference is also insignificant, this could be possible because when the point cloud pipeline is making the Z-test and the particle is discarded it

does not draw that point on the screen and it does not need to update the Z value, saving time.

Seems that Z-test is affecting negatively the times of OpenGL pipeline but it is not penalizing so much the cloud point pipeline.

6.0.1.3 Dataset 3

The third dataset consists on 700.000 randomly created vertices with particle size 3. The objective of creating this dataset is to see the impact of the size of the dataset in the different pipelines. Since the previous datasets have processed a small amount of vertices. Now, the number of vertices that both pipelines are going to process is higher and higher times are expected. In the Figure 6.3 we can observe the results of this test.

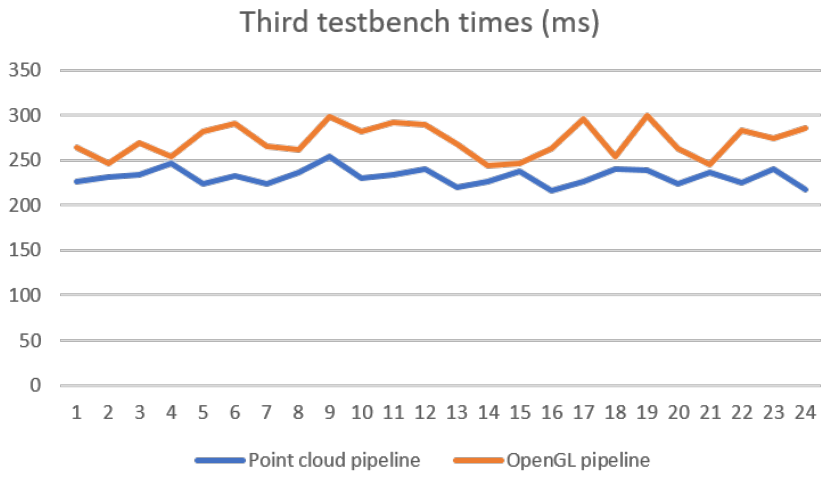

Figure 6.3: Plot of the different times of the third dataset

As we can observe the OpenGL pipeline seems to process the information a little bit slower compared to the Point cloud pipeline, this would mean that the size of the data seems to affect more negatively the OpenGL pipeline than the point cloud pipeline. The average time of the OpenGL pipeline is of 271.08 ms and the point cloud pipeline of 231.78 ms. The speed up of the point cloud pipeline respect the OpenGL pipelin is of 1.17.

6.0.1.4 Dataset 4

The fourth dataset is large dataset, it contains 1 milion of vertices created randomly and with particle size 9, expecting lot of Z-test conflicts. With this test we want to find out what happens when we increase the number of vertices and moreover, we increase the number of Z-test conflicts. As we have seen in other datasetes as more Z-test done, the OpenGL seems to answer slower than our pipeline. On the other hand, with small datasets and without Z-tests seems to work faster, but these conditions seem to be not characteristic of real-data.

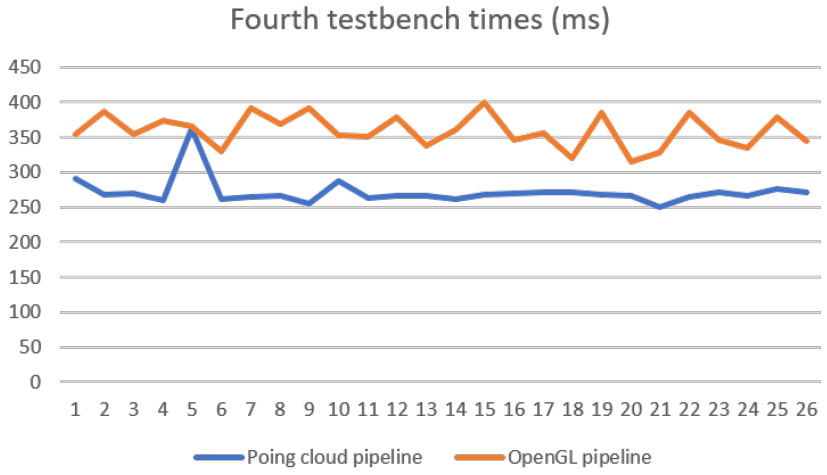

Figure 6.4: Plot of the different times of the fourth dataset

As we can observe in 6.4, the point cloud pipeline is processing the information faster than the OpenGL pipeline, the optimizations and the deletions of useless information for point primitive seem to work better with large data for the point cloud pipeline. In this plot we can see a very significant difference of time between each other, the average time of the point cloud pipeline is of 271.72 ms and the average time of the OpenGL pipeline is of 359.17 ms. In this dataset the speed up of the point cloud respect the OpenGL pipeline is of 1.321.

6.0.2 Real data

In this section we will compare the times between the point cloud pipeline and the OpenGL pipeline with real scientific data. The scientific data has been obtained from Illustris project, this data consists on 292972 vertices, in this case we suppose the particle size as 0, so there are not neighbour subparticles. In Figure 6.5 we can see the results of this dataset.

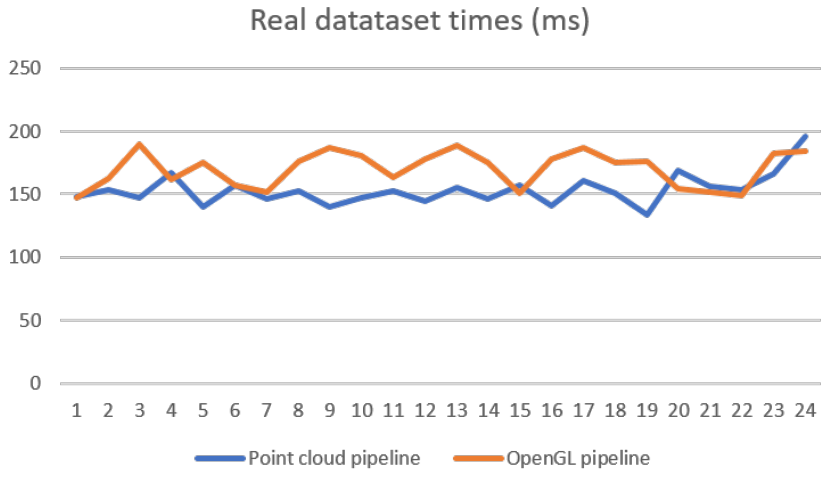

Figure 6.5: Plot of the different times of the Illustrist Project

As we can observe in the plot, the point cloud pipeline seems to process a little bit faster the information than the OpenGL pipeline in this dataset, the average time for the point cloud pipeline is of 153.522 ms and the OpenGL pipeline is of 170.16 ms. The speed up of the point cloud pipeline respect the OpenGL pipeline is of 1.11. The difference between them seem to not be large.

In the next Figure 6.6, we can observe part of the rendering of the Illustris Project with the point cloud pipeline.

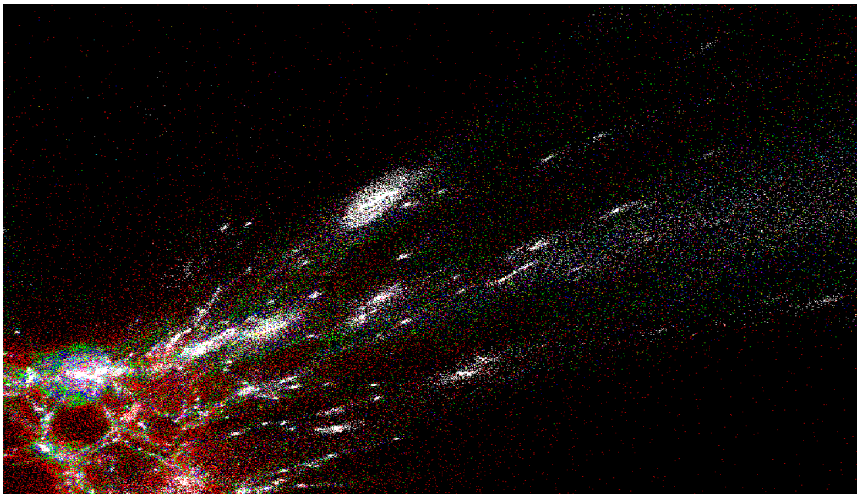

Figure 6.6: Picture of the rendering of Illustris Project data with the cloud point pipeline

As we have seen in this and previous datasets, the point cloud pipeline seems to work faster with a large quantity of vertices than the OpenGL pipeline, on the other hand, OpenGL

pipeline seems to work faster with small quantity of vertices. Moreover, there has been a determining factor agains the point cloud pipeline, and this was the attribute of particle size, since the process of adding neighbourhood points it is not parallelized, it is done in thesame thread of the initial subparticle.

7. Conclusion

As we have seen in the previous chapter, we were able to build a point cloud pipeline that is faster than the OpenGL pipeline in some cases. The speed up obtain in those cases was not huge, but enough to demonstrate that there are chances to build faster pipelines for this kind of large and unstructured data. Although this, there are some cases where the OpenGL pipeline was faster and this would be a good research to follow, how to improve this cases. As I mentioned before, there was a determining fact that slow down the point cloud pipeline, this was the particle size, that the drawing of the subparticles was programmed in a sequential way inside the thread of the particle. For this reason, the same dataset with 0 particle size and 9 particle size varies the time.

Some results of the test were surprising, for example, the point cloud pipeline seems to not increase a lot the times when the data is big, but on the other hand, when we cope a big data with a big particle size, the time seems to increase a lot.

Another fact that could have influenced the results is the processor and the GPU, we can not obtain the same results with another computer, since the hardware is different. Even do with the same hardware, we can't obtain the same times because the computer is under certain external conditions, such as independent processes opened at the same time that consumes resources.

Another aspect to improve is the flickering provoked by the Z-fighting. In some cases when we have particles that have the same x y z coordinates but different color, there is a situation that provokes colour flickering. This situation can be shorten deleting one of this particles or adding resolution to the depth buffer. On the other hand, the most used manner solution is using the stencil buffer.

Another aspect that can improve the speed up of the point cloud pipeline is adding some optimization techniques like Z-early that can improve the average times of the point cloud pipeline. Another technique that can be useful for this project is the preprocessing of point clouds, dealing with data size.

Bibliography

[AMD] "Amd opencl developer," http://www.amd.com/es-xl/solutions/professional/hpc/opencl.

[GKLR13] C. Günther, T. Kanzok, L. Linsen, and P. Rosenthal, "A gpgpu-based pipeline for accelerated rendering of point clouds," 2013.

[HA11] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," pp. 78–90, 2011.

[int] "Intel opencl developer," https://software.intel.com/en-us/intel-opencl.

[LW85] M. Levoy and T. Whitted, *The use of points as a display primitive.* University of North Carolina, Department of Computer Science, 1985.

[nvi] "Nvidia opencl developer," https://developer.nvidia.com/opencl.

[op]

[Sca12] M. Scarpiano, *OpenCL in Action:How to Accelerate Graphics and Computation*, 2012.

[Tay13] R. Tay, *OpenCL Parallel Programming Development Cookbook*, 2013.

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den September 18, 2017

(Albert Pérez Toro)